

# Spring

---

## 6: Spring MVC



### ÍNDICE

1	Introducción .....	2
2	Primer controlador y ejemplo de vista.....	7
3	Procesando parámetros vía GET URL.....	10
4	Trabajando con Formularios. ....	11
5	Validación de Formularios.....	15
6	Conexión a MySQL con Jdbc Template.....	19
7	CRUD con MySQL y Jdbc Template.....	22

# 1 Introducción

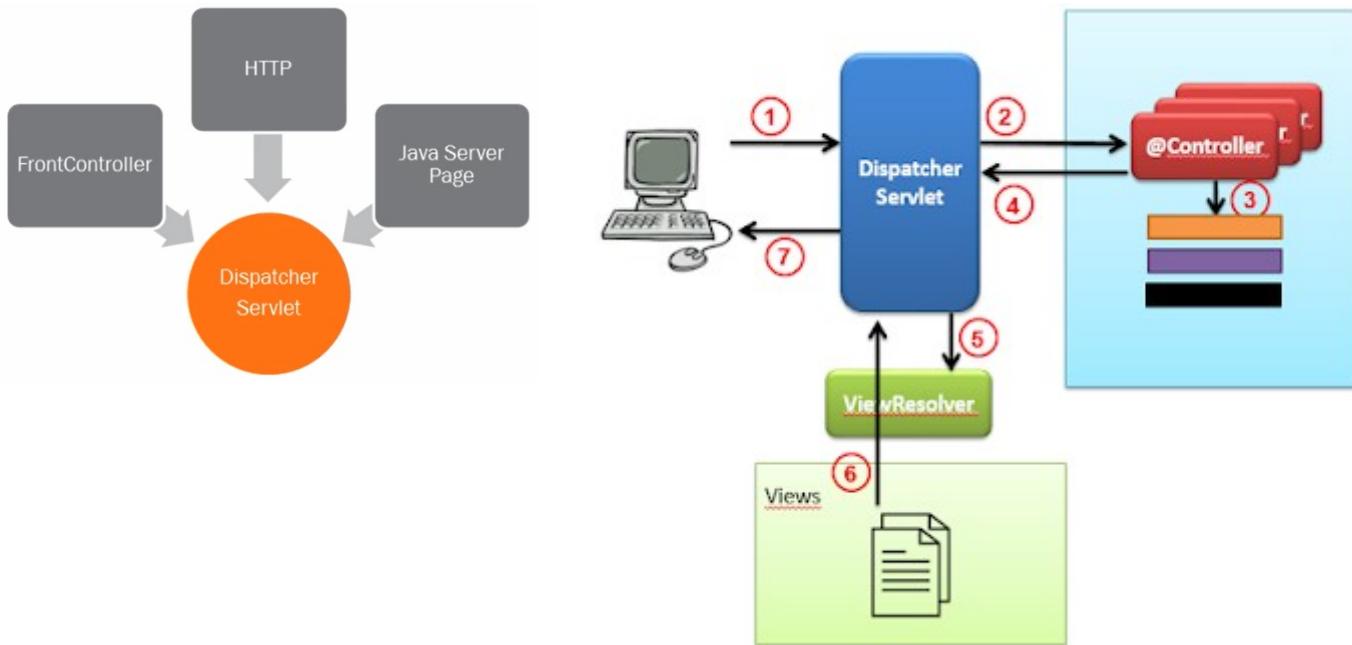
Qué es Spring.

- Es un Framework de Java Platform Enterprise Edition (Java EE).
- Rod Johnson lo lanzó junto a la publicación de su libro *Expert One-on-One J2EE Design and Development* (Wrox Press, octubre 2002).
- Se basa en el patrón Modelo Vista Controlador.
- Permite crear aplicaciones robustas y basadas en convenciones.
- Unifica distintas APIs de gestión y coordina las transacciones para los objetos de Java.
- Permite programación Orientada a Aspectos.

Spring Web MVC es un sub-proyecto Spring que está dirigido a facilitar y optimizar el proceso creación de aplicaciones web utilizando el patrón MVC (**Modelo-Vista-Controlador**), donde el **Modelo** representa los datos o información que manejará la aplicación web, la **Vista** son todos los elementos de la UI (Interfaz de Usuario), con ellos el usuario interactúa con la aplicación, ejemplo: botones, campos de texto, etc., finalmente el **Controlador** será el encargado manipular los datos en base a la interacción del usuario.

La pieza central de Spring MVC es un componente llamado el "**DispatcherServlet**", el cual sigue el patrón de diseño "**Front controller**", este envía las peticiones a los componentes designados para manejarlas.

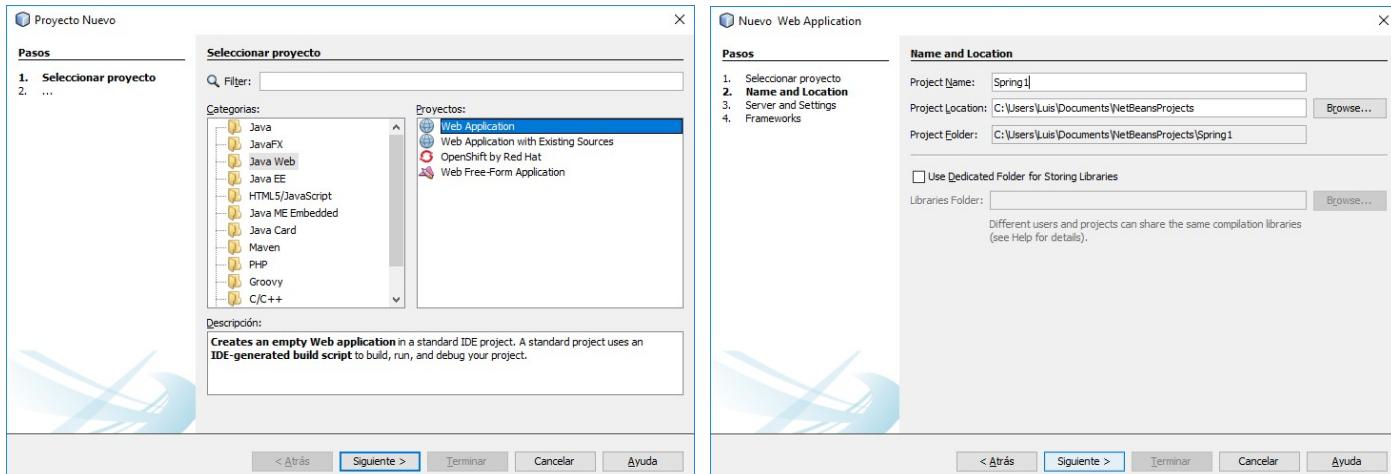
El patrón de diseño "**Front controller**" aparece en varios catálogos de patrones y está relacionado con el diseño de aplicaciones web. Es "un controlador que maneja todas las solicitudes de un sitio web", que es una estructura útil para que los desarrolladores de aplicaciones web logren la flexibilidad y la reutilización sin redundancia de código.



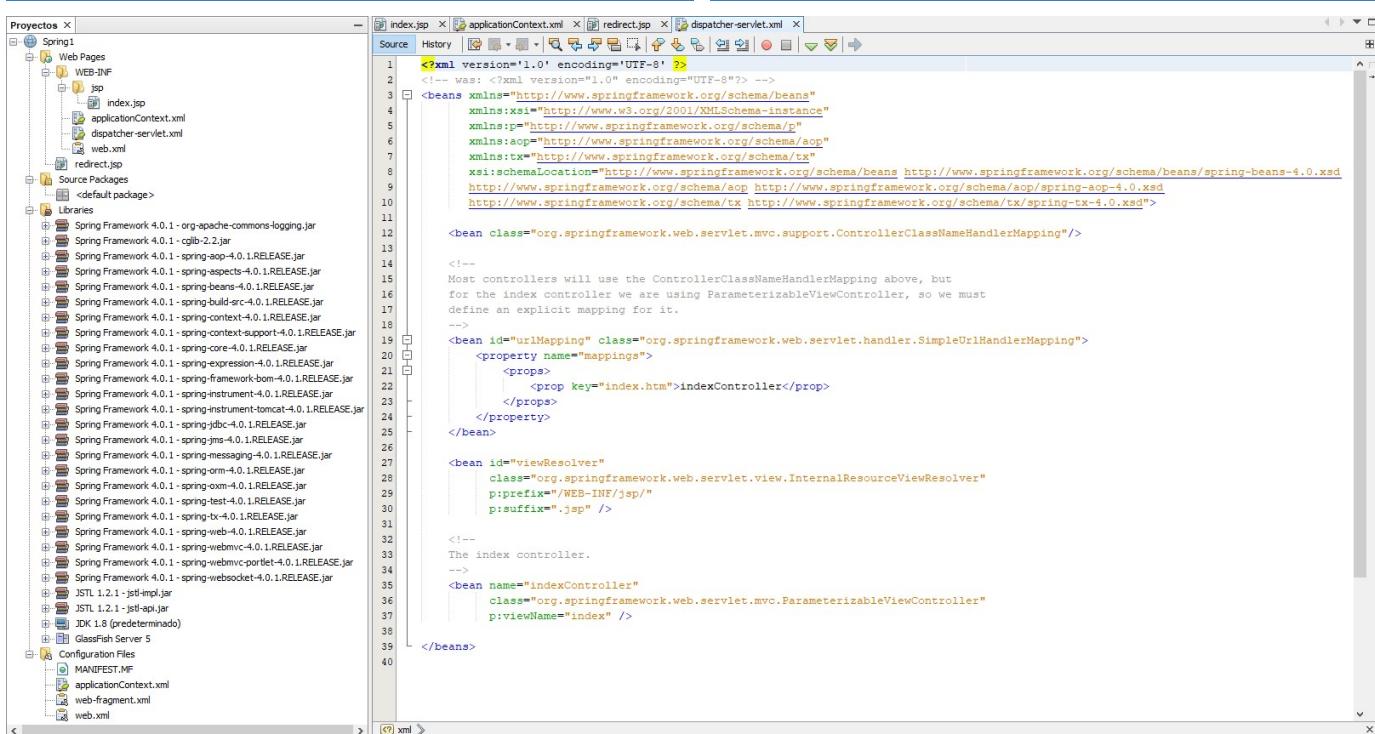
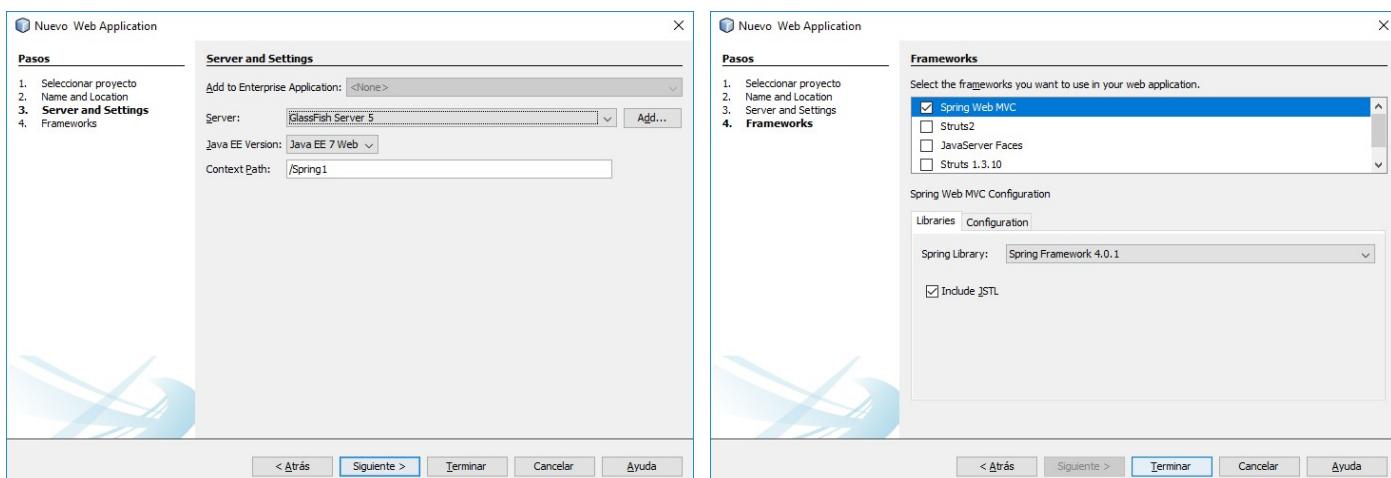
Funcionamiento básico del **DispatcherServlet**:

1. El cliente hace una petición a la aplicación web, esta petición llega al **DispatcherServlet**.
2. El **DispatcherServlet** determina qué componente debe atender la petición y la envía a este.
3. El **Controller** implementa la lógica específica para responder la petición, para lo que puede hacer uso de cualquier recurso que esté al alcance de cualquier aplicación Java (incluyendo conexiones con servicios web o con base de datos).
4. Una vez que el **Controller** termina su proceso, regresa la petición al **DispatcherServlet**, estableciendo los datos adecuados del modelo e indicando el nombre lógico de la vista que debe regresarse al cliente y un modelo lleno con los nombres y valores de los atributos que se usarán para generar la vista final.
5. En base al nombre lógico regresado por el **Controller**, el **DispatcherServlet** usa un **ViewResolver** para determinar qué recurso debe utilizar para generar la vista final que se mostrará al usuario, este recurso puede ser una **JSP**, una página HTML, un template de Velocity, un archivo de Excel, un PDF, etc.
6. El **DispatcherServlet** obtiene la vista que será regresada al cliente.
7. El **DispatcherServlet** finalmente regresa la vista adecuada al cliente.

Comenzaremos a crear un proyecto web como se ha visto hasta ahora. “Nuevo Proyecto”, Categoría “Java Web”, y “Web Application”, con el nombre “Spring1”.



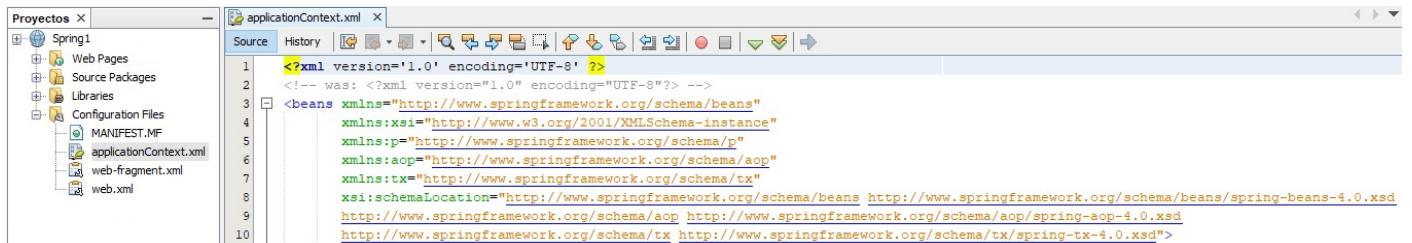
Elegimos de servidor “GlassFish Server 5” y la novedad viene en la pantalla de **Frameworks**, que marcamos la primera casilla, “Spring Web MVC”, y dejamos marcada la casilla de “Include JSTL”, para que incluya también estas librerías.



Al crearse el proyecto se muestran varios archivos y tenemos cuatro directorios principales.



En “**Configuration Files**” hay archivos de configuración, siendo el más importante “**application Context.xml**”, que es el que usa el Framework para hacer algunas referencias a los namespaces que necesita para cargar.



```

<?xml version='1.0' encoding='UTF-8'?>
<!-- was: <?xml version='1.0' encoding='UTF-8'?> -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
                           http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
                           http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">

```

Luego tiene la configuración de las “**properties**”.

```

<!--bean id="propertyConfigurer"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
      p:location="/WEB-INF/jdbc.properties" />

```

Y después la creación de “**dataSource**”, para trabajar con algún motor de persistencia.

```

<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      p:driverClassName="${jdbc.driverClassName}"
      p:url="${jdbc.url}"
      p:username="${jdbc.username}"
      p:password="${jdbc.password}" /-->

<!-- ADD PERSISTENCE SUPPORT HERE (jpa, hibernate, etc) -->
</beans>

```

Otro fichero destacado es el “**web.xml**”.



```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.w3.org/2001/XMLSchema-instance contextConfigLocation=WEB-INF/applicationContext.xml">
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/applicationContext.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>2</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
</web-app>

```

Podemos definir que archivo se mostrará cuando se llame desde el navegador a la carpeta del proyecto solamente.

```

<welcome-file-list>
    <welcome-file>redirect.jsp</welcome-file>
</welcome-file-list>

```

En “**Libraries**” están cargadas todas las librerías necesarias para trabajar con Spring.



```

<!--
    Most controllers will use the ControllerClassNameHandlerMapping above, but
    for the index controller we are using ParameterizableViewController, so we must
    define an explicit mapping for it.
-->
<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="index.htm">indexController</prop>
        </props>
    </property>
</bean>

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix">/WEB-INF/jsp/</property>
    <property name="suffix">.jsp</property>
</bean>

<!--
    The index controller.
-->
<bean name="indexController" class="org.springframework.web.servlet.mvc.ParameterizableViewController">
    <property name="viewName">index</property>
</bean>

```

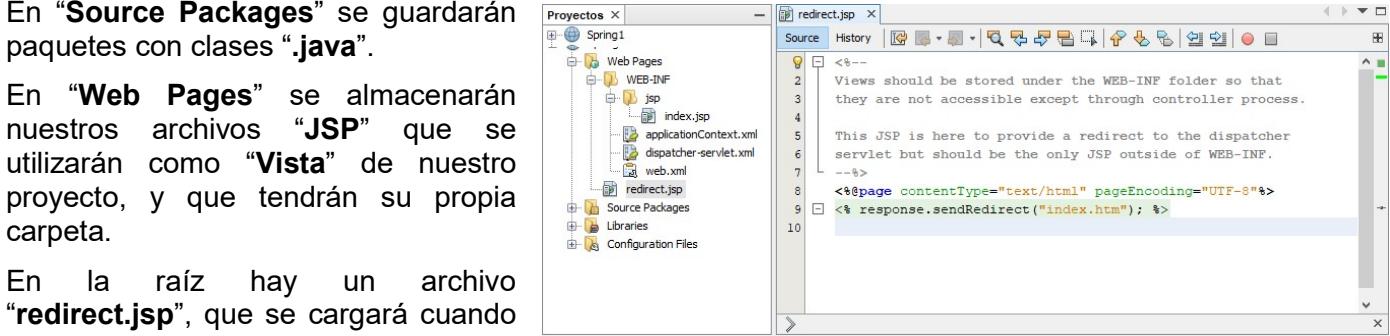
En “Source Packages” se guardarán paquetes con clases “.java”.

En “Web Pages” se almacenarán nuestros archivos “JSP” que se utilizarán como “Vista” de nuestro proyecto, y que tendrán su propia carpeta.

En la raíz hay un archivo “**redirect.jsp**”, que se cargará cuando en el navegador tan solo se ponga el nombre del proyecto.

Con “**redirect.jsp**”, se ejecutará un redireccionamiento a nivel de servidor a “**index.htm**”.

El archivo “**dispatcher-servlet.xml**” permite que a través de él le informemos al Framework de cada uno de los recursos de navegación que dispongamos. Un recurso de navegación es básicamente cada página que nosotros creemos. Este archivo, es el que utiliza Spring como si fuera un servlet general, que está preparado para poder recibir todas las peticiones y canalizarlas a través de este mismo archivo. De esta forma, para poder navegar entre las distintas páginas, o poder acceder a sus funcionalidades, tendremos que indicarlas aquí.



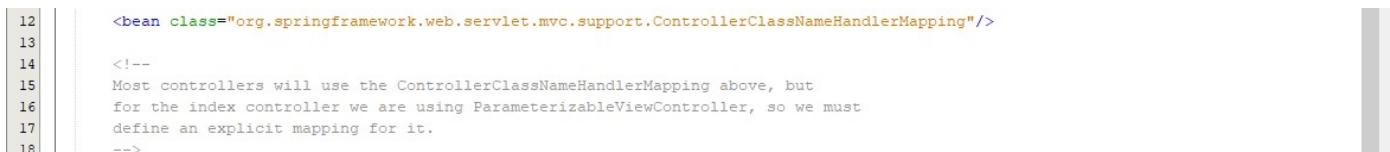
```

<%-- Views should be stored under the WEB-INF folder so that they are not accessible except through controller process.
This JSP is here to provide a redirect to the dispatcher servlet but should be the only JSP outside of WEB-INF.
--%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<% response.sendRedirect("index.htm"); %>

```

Si no tenemos cargado ningún controlador, como es el caso nuestro, porqué todavía no se ha realizado nada, se va a cargar un controlador por defecto que se llama “**Front Controller**”.

Mediante los elementos “**<bean>**” se van a ir llamando a distintos recursos, por ejemplo el “**ControllerClassNameHandlerMapping**”, que va a permitir canalizar todas las peticiones que se van a realizar a través de la URL.



```

<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

<!--
Most controllers will use the ControllerClassNameHandlerMapping above, but
for the index controller we are using ParameterizableViewController, so we must
define an explicit mapping for it.
-->

```

Después, hay otro “**<bean>**”, el “**urlMapping**”, que tiene preconfigurado todo lo necesario para que el Framework entienda que dentro de las etiquetas “**<property>**”, podemos ir creando etiquetas de “**<props>**” para ir informando al Framework de cada uno de los recursos que nosotros vayamos creando. En nuestro caso, ya hay un recurso creado por defecto, dentro de la etiqueta “**<prop>**”, donde se indica el nombre de la petición o de la página que se llamará desde el navegador, y que llama al controlador “**indexController**”, creado por defecto por NetBeans.



```

<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="index.htm">indexController</prop>
        </props>
    </property>
</bean>

```

Para gestionar las vistas, utilizamos otro “**<bean>**” más, el “**viewResolver**”, que además de la ruta de la clase, tendrá la ubicación donde buscar los ficheros, “**/WEB-INF/jsp/**”, y la extensión de ficheros a utilizar, “**.jsp**”.



```

<bean id="viewResolver"
    class="org.springframework.web.servlet.view.InternalResourceViewResolver"
    p:prefix="/WEB-INF/jsp/"
    p:suffix=".jsp" />

```

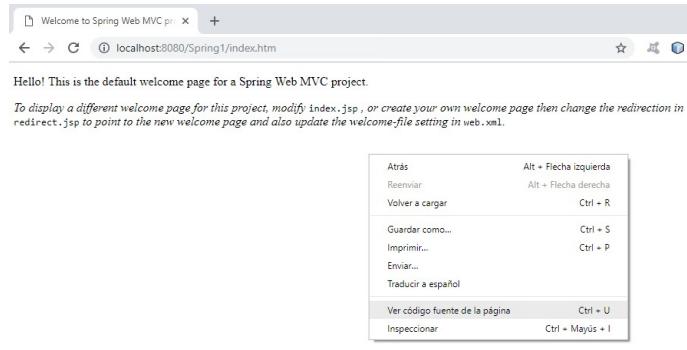
Cada controlador creado entre las etiquetas “**<props>**”, tendrá más abajo, otro elemento “**<bean>**”, como el de nombre “**indexController**”, con la ruta de la clase y el valor “**index**”, que hace referencia al fichero “**index.jsp**”.

```

35 <bean name="indexController"
36     class="org.springframework.web.servlet.mvc.ParameterizableViewController"
37     p:viewName="index" />
38
39 </beans>

```

Al ejecutar el proyecto podemos comprobar que funciona correctamente, apareciendo el texto de bienvenida que se encuentra en el fichero “**index.jsp**” y que es llamado desde el recurso “**index.htm**”. También podemos ver el código fuente de la página, totalmente en html.



```

1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
2   "http://www.w3.org/TR/html4/loose.dtd">
3
4 <html>
5   <head>
6     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7     <title>Welcome to Spring Web MVC project</title>
8   </head>
9
10  <body>
11    <p>Hello! This is the default welcome page for a Spring Web MVC project.</p>
12    <p><i>To display a different welcome page for this project, modify</i>
13      <tt>index.jsp</tt> <i>, or create your own welcome page then change
14        the redirection in</i> <tt>redirect.jsp</tt> <i>to point to the new
15          welcome page and also update the welcome-file setting in</i>
16            <tt>web.xml</tt>.</p>
17
18  </body>
19 </html>

```

Si en el navegador ponemos solamente el nombre del proyecto, sin indicarle ningún fichero ni recurso.

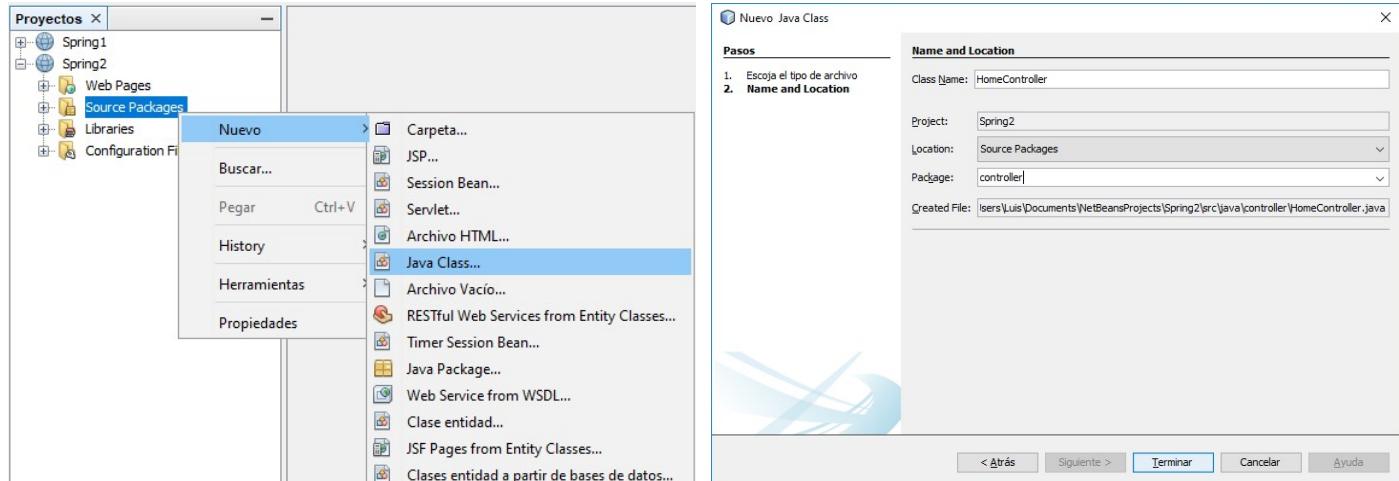
En el fichero “**web.xml**” tiene configurado que se inicie el fichero “**redirect.jsp**”, y como éste tiene en su contenido un redireccionamiento a “**index.htm**”, al ejecutarse este recurso desde el “**dispatcher-servlet.xml**”, volveríamos al caso anterior, ejecutándose exactamente igual que antes.

## 2 Primer controlador y ejemplo de vista.

Continuamos desde el proyecto anterior, cerrando todos sus archivos, seleccionando el nombre, y desde el menú contextual, elegimos “Copiar”. Podemos de nombre “Spring2” y pulsamos el botón “Copiar”.

Ahora ya podemos cerrar todos los archivos de “Spring1” y contraer sus ramas de directorios. Vamos a realizar algunas modificaciones en “Spring2” para aclarar los conceptos vistos hasta ahora.

Señalamos “Source Package”, con botón derecho “New” y “Java Class” para crear un controlador, de nombre “HomeController”, en el paquete “controller”. Es recomendable que se ponga este nombre del paquete para recoger todos los controladores, y el nombre, también conviene que termine en Controller.



Ahora que ya tenemos la clase creada, vamos a darle la estructura para que el Framework pueda entender que ésta clase va a ser utilizada como un controlador.

Borramos los comentarios, creamos un método llamado “home” de tipo “ModelAndView”, para manejar el modelamiento de vistas desde un controlador al estilo de Spring. Tenemos que importar ésta clase, eligiendo la correcta, y no la primera sin fijarse, “org.springframework.web.servlet.ModelAndView”.

A screenshot of the NetBeans IDE showing the 'HomeController.java' file. The code is as follows:

```
package controller;
import org.springframework.web.servlet.ModelAndView;
public class HomeController {
    public ModelAndView home() {
    }
}
```

The code editor has tabs for 'Source' and 'History'. The 'Source' tab is active, showing the code. The 'History' tab is also visible. The code editor has various toolbars and icons at the top.

Vamos a darle al controlador cierta información, mediante una anotación “@Controller” que colocamos justo antes de declarar la clase. De esta forma, se informará que es un controller, un controlador, y tenemos que importar una clase, que será la primera opción propuesta del menú contextual.

A screenshot of the NetBeans IDE showing the 'HomeController.java' file. The code now includes the '@Controller' annotation:

```
package controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
@Controller
public class HomeController {
    @RequestMapping("home.htm")
    public ModelAndView home() {
    }
}
```

The '@Controller' annotation is highlighted in yellow. The code editor has tabs for 'Source' and 'History'. The 'Source' tab is active, showing the code. The 'History' tab is also visible. The code editor has various toolbars and icons at the top.

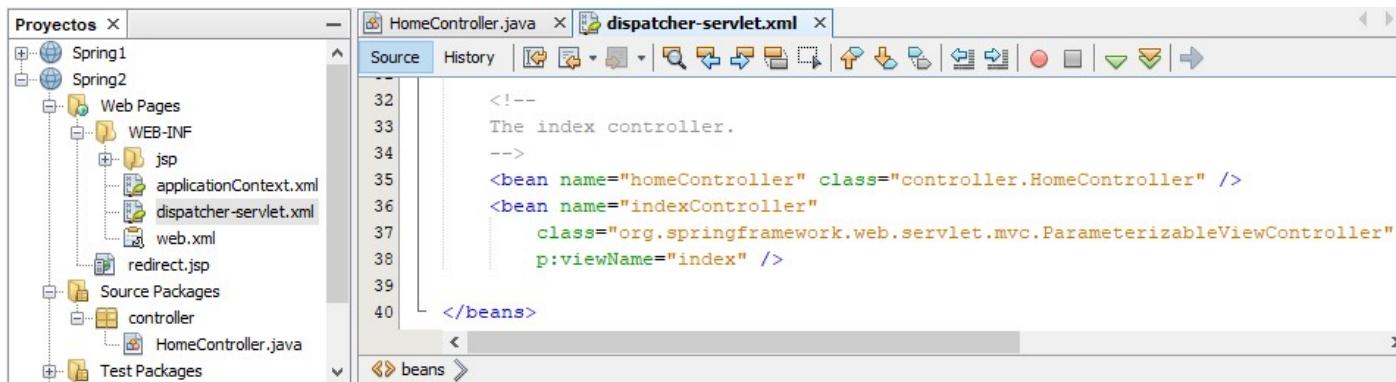
Dentro del método, creamos una instancia de “ModelAndView” llamada “mav”. Después utilizamos “mav.setViewName”, para indicarle el nombre de la vista asociada, que será “home”. Por último, utilizamos “return mav” para devolver la variable “mav”. De esta manera, el Framework sabe que éste metodo se llamará cuando se indique en la URL “home.htm”.

Mediante otra anotación “@RequestMapping”, informaremos que el método que estamos utilizando va a ser utilizado como una Vista. De esta forma, cada uno de los métodos que incorporemos al controlador, debería representar una página. Necesitamos importar otra clase, que será la primera opción propuesta del menú contextual. A esta anotación, le pasamos como parámetro la referencia que nosotros vamos a utilizar para pedir éste método en la URL, quedando “@RequestMapping(“home.htm”)”.

```
@RequestMapping("home.htm")
public ModelAndView home() {
    ModelAndView mav = new ModelAndView();
    mav.setViewName("home");
    return mav;
}
```

Ahora que el controlador está listo, para que funcione correctamente, tenemos que informar sobre este controlador en el “**dispatcher-servlet.xml**”. Recordemos que se encuentra dentro de “**Web Pages**” y de “**WEB-INF**”.

Primero nos iremos al final del fichero para crear un objeto de “**<beans>**”. Con el parámetro “**name**” le indicamos el nombre de nuestro controlador, “**homeController**” y con el atributo “**class**” le indicamos su ruta, “**controller.HomeController**”.



```

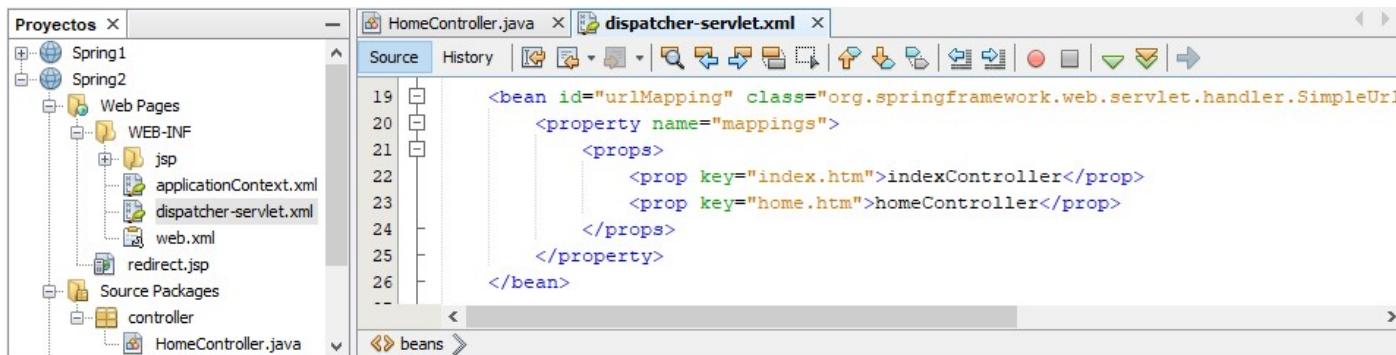
<!--
The index controller.
-->
<bean name="homeController" class="controller.HomeController" />
<bean name="indexController"
      class="org.springframework.web.servlet.mvc.ParameterizableViewController"
      p:viewName="index" />

</beans>

```

De esta manera, el Framework se dará por enterado de que tenemos creado este controlador. La idea es realizar lo mismo por cada controlador que necesitemos crear.

En segundo lugar , más arriba, entre las etiquetas “**<props>**”, se informará de las Vistas que tenemos creadas. Por lo tanto, para nuestra vista que hemos creado, introducimos una etiqueta “**<prop>**”, y con el parámetro “**key**” le indicamos que para “**home.htm**” se busque ese recurso en el controlador “**homeController**”.

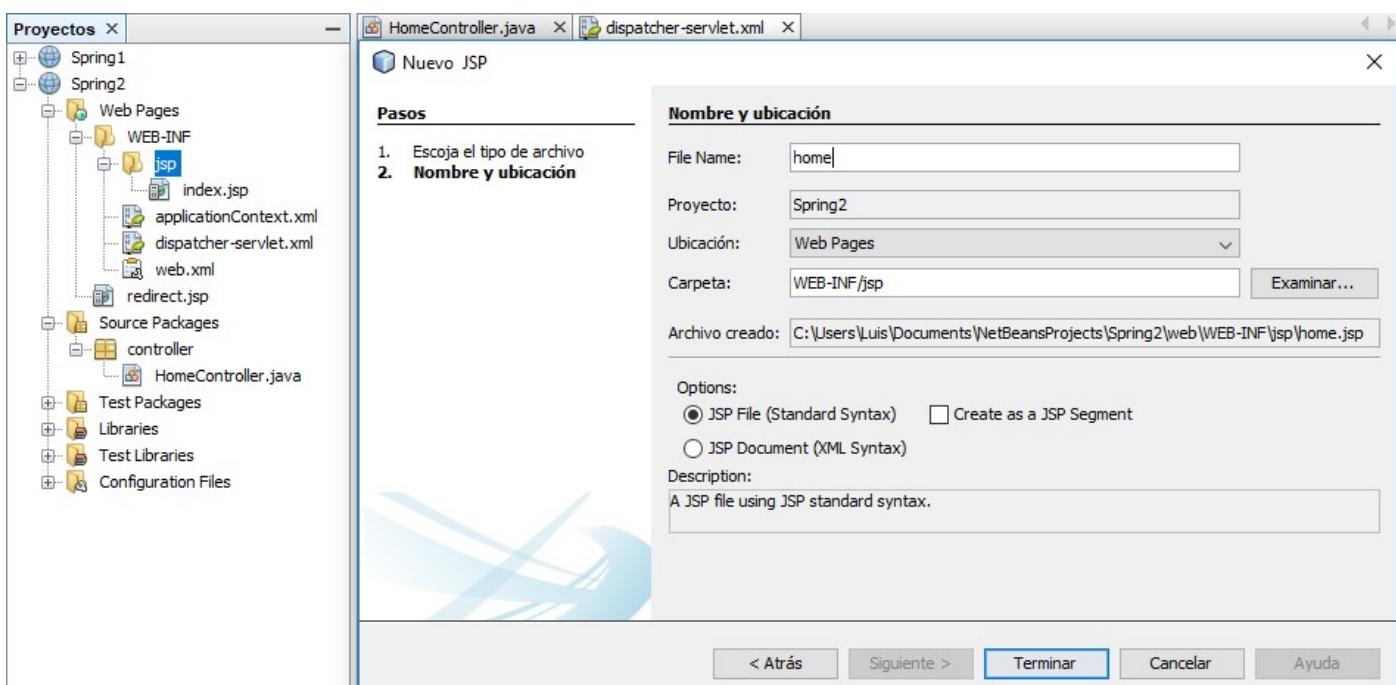


```

<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="index.htm">indexController</prop>
      <prop key="home.htm">homeController</prop>
    </props>
  </property>
</bean>

```

Para terminar, creamos el fichero “**home.jsp**” que tiene que soportar ésta vista, en “**Web Pages**” y “**JSP**”.



Una vez creado el fichero, lo modificamos, quitando los comentarios, poniendo de título “**Ejemplo de Vista Spring**” y en el “**<h1>**”, “**Home**”. En la ejecución, cambiamos en la URL “**index.htm**” por “**home.htm**”.

```

1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <!DOCTYPE html>
3 <html>
4   <head>
5     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6     <title>Ejemplo de Vista Spring</title>
7   </head>
8   <body>
9     <h1>Home</h1>
10    </body>
11 </html>

```

Vamos a crear otro método, y para ello, copiamos el que ya tenemos, cambiando “home”, por “contacto”

```

1 package controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.servlet.ModelAndView;
6
7 @Controller
8 public class HomeController {
9
10   @RequestMapping("home.htm")
11   public ModelAndView home(){
12
13     ModelAndView mav = new ModelAndView();
14     mav.setViewName("home");
15     return mav;
16   }
17
18   @RequestMapping("home.htm")
19   public ModelAndView contacto(){
20
21     ModelAndView mav = new ModelAndView();
22     mav.setViewName("home");
23     return mav;
24   }
25 }

```

```

1 package controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.servlet.ModelAndView;
6
7 @Controller
8 public class HomeController {
9
10   @RequestMapping("home.htm")
11   public ModelAndView home(){
12
13     ModelAndView mav = new ModelAndView();
14     mav.setViewName("home");
15     return mav;
16   }
17
18   @RequestMapping("contacto.htm")
19   public ModelAndView contacto(){
20
21     ModelAndView mav = new ModelAndView();
22     mav.setViewName("contacto");
23     return mav;
24   }
25 }

```

Ahora tenemos que informar en el “dispatcher-servlet.xml”. Como ya tenemos el “<bean>” al final del fichero del controlador, solo nos falta añadir un “<prop>”, con el parámetro “key” de valor “contacto.htm”, para que busque ese recurso también en el controlador “HomeController”. Por último, nos falta crear la vista en JSP “contacto.jsp”, copiandolo de “home.jsp” y cambiando el “<h1>” por “Contacto”. Hecho esto, lo podemos probar.

```

19 <bean id="urlMapping" class="org.springframework.web.servlet.
20   <property name="mappings">
21     <props>
22       <prop key="index.htm">indexController</prop>
23       <prop key="home.htm">homeController</prop>
24       <prop key="contacto.htm">homeController</prop>
25     </props>
26   </property>
27 </bean>

```

```

1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <!DOCTYPE html>
3 <html>
4   <head>
5     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6     <title>Ejemplo de Vista Spring</title>
7   </head>
8   <body>
9     <h1>Contacto</h1>
10    </body>
11 </html>

```

Cambiamos el contenido de “index.jsp”, copiándolo de otro JSP, y ponemos de “<h1>”, “Index”.

```

1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <!DOCTYPE html>
3 <html>
4   <head>
5     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6     <title>Ejemplo de Vista Spring</title>
7   </head>
8   <body>
9     <h1>Index</h1>
10    </body>
11 </html>

```

Ejemplo de Vista Spring  
localhost:8080/Spring2/home.htm

Home

Ejemplo de Vista Spring  
localhost:8080/Spring2/contacto.htm

Contacto

Ejemplo de Vista Spring  
localhost:8080/Spring2/index.htm

Index

### 3 Procesando parámetros vía GET URL.

Continuamos desde el proyecto anterior, “Spring2”, cerrando todos sus archivos, seleccionando el nombre, y desde el menú contextual, “Copiar”, de nombre “Spring3” y pulsamos el botón “Copiar”.

Ahora ya podemos cerrar “Spring2” y realizar algunas modificaciones en “Spring3”.

Vamos ver cómo recibir parámetros desde la URL, por ejemplo, “localhost:8080/Spring3/home.htm?id=1”.

Una de las formas más utilizadas para realizar esto, es utilizando la clase “**HttpServletRequest**”, de tal manera que a cada uno de los métodos que quieras procesar parámetros que vengan a través de la URL, vía GET, como parámetro se le va a pasar un objeto de tipo “**HttpServletRequest**”, importar su clase relacionada, y ponerle un nombre, como por ejemplo, “request”.

Declaramos dentro del método cada variable que vayamos a utilizar con “**request.getParameter(“<Variable>”)**” para recuperar el valor.

Pasamos cada valor del controlador a la vista con “**mav.addObject(“<Variable>”)**”,

Pasamos cada valor del controlador a la vista de forma que, a cada instancia de la clase  **ModelAndView**, utilizaremos el atributo “ **addObject(String attributeName, Object attributeValue)**”.

```
Proyectos X Spring3
Web Pages
  WEB-INF
    jsp
      contacto.jsp
      home.jsp
      index.jsp
      applicationContext.xml
      dispatcher-servlet.xml
      web.xml
    redirect.jsp
  Source Packages
    controller
      HomeController.java
  Test Packages
  Libraries
  Test Libraries
  Configuration Files

HomeController.java X
Source History
1 package controller;
2
3 import javax.servlet.http.HttpServletRequest;
4 import org.springframework.stereotype.Controller;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.servlet.ModelAndView;
7
8 @Controller
9 public class HomeController {
10
11     @RequestMapping("home.htm")
12     public ModelAndView home( HttpServletRequest request) {
13
14         ModelAndView mav = new ModelAndView();
15         mav.setViewName("home");
16         String id = request.getParameter("id");
17         String nombre = request.getParameter("nombre");
18         mav.addObject("id", id);
19         mav.addObject("nombre", nombre);
20
21     }
}
```

Ahora en la vista, modificamos el fichero “**home.jsp**”.

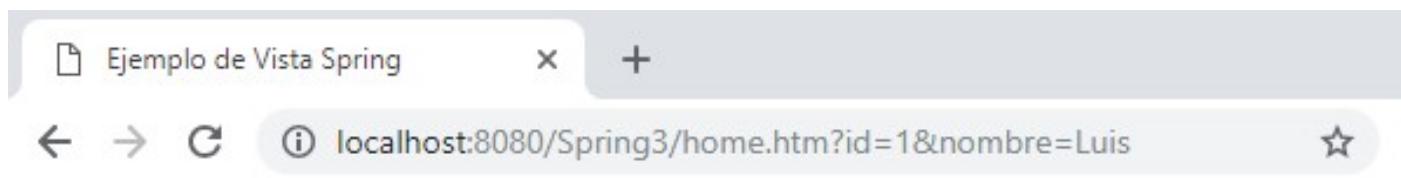
Incluimos los “**taglib**” para usar las etiquetas comunes de JSTL.

Dentro de las etiquetas de lista, mostramos los valores de los dos argumentos pasados por URL usando “**<c:out>**”.

Probamos la ejecución.

```
Proyectos X Spring3
Web Pages
  WEB-INF
    jsp
      contacto.jsp
      home.jsp
      index.jsp
      applicationContext.xml
      dispatcher-servlet.xml
      web.xml
    redirect.jsp
  Source Packages
    controller
      HomeController.java
  Test Packages
  Libraries
  Test Libraries
  Configuration Files

home.jsp X
Source History
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <!DOCTYPE html>
4 <html>
5   <head>
6     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7     <title>Ejemplo de Vista Spring</title>
8   </head>
9   <body>
10    <h1>Home</h1>
11    <ul>
12      <li>ID = <c:out value="${id}" /></li>
13      <li>NOMBRE = <c:out value="${nombre}" /></li>
14    </ul>
15  </body>
16 </html>
```



## Home

- ID = 1
- NOMBRE = Luis

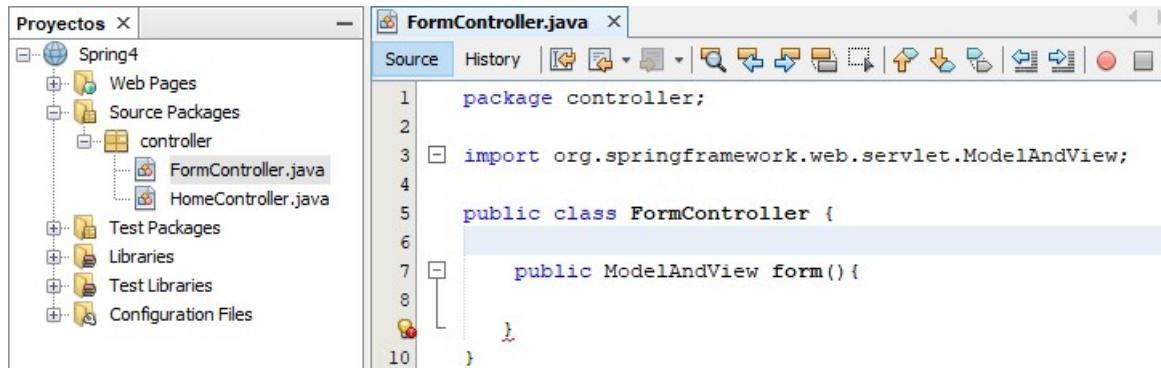
## 4 Trabajando con Formularios.

Continuamos desde el proyecto anterior, “Spring3”, cerrando todos sus archivos, seleccionando el nombre, y desde el menú contextual, “Copiar”, de nombre “Spring4” y pulsamos el botón “Copiar”.

Ahora ya podemos cerrar “Spring3” y realizar modificaciones en “Spring4”.

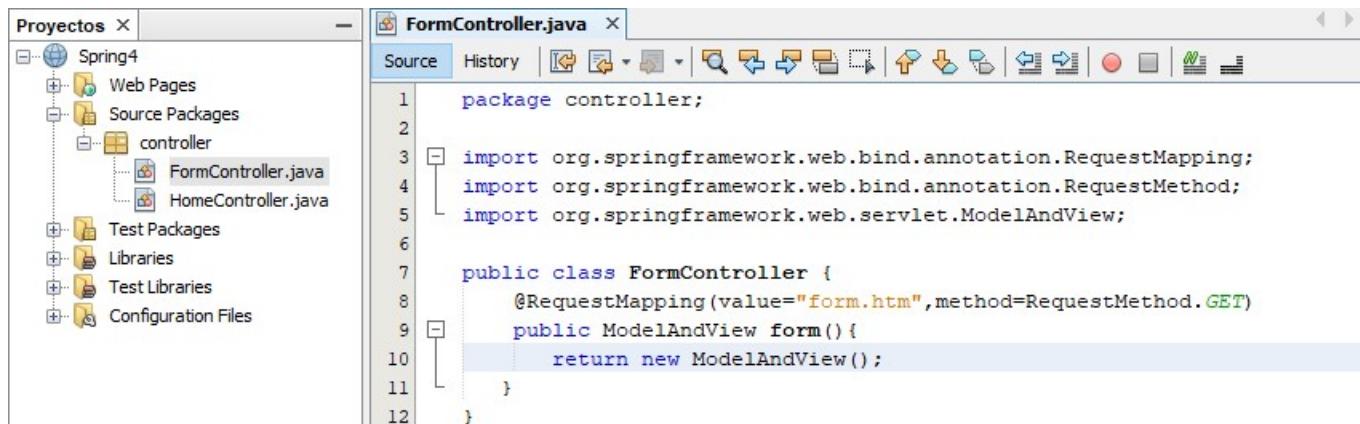
Para generar nuestro formulario, vamos a crear un nuevo controlador. Señalamos “Source Package”, con botón derecho “New” y “Java Class” para crear un controlador, de nombre “FormController”, en el paquete “controller”.

Borramos los comentarios, creamos un método llamado “form” de tipo “ModelAndView”, e importamos la clase, “org.springframework.web.servlet. ModelAndView”.



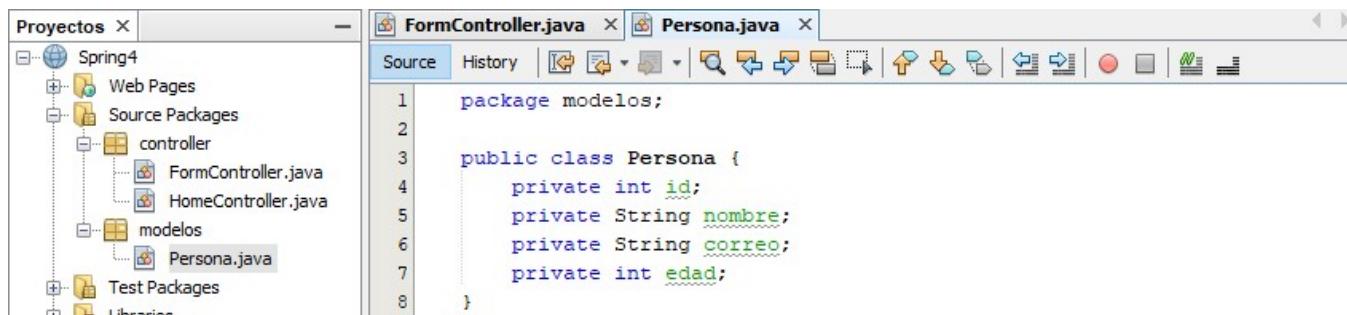
```
package controller;
import org.springframework.web.servlet.ModelAndView;
public class FormController {
    public ModelAndView form() {
    }
}
```

Con la anotación “@RequestMapping”, informaremos que el método que estamos utilizando va a ser utilizado como una Vista, e importamos su clase relacionada. Le pasamos como parámetro el atributo “value”, con la referencia que nosotros vamos a utilizar para pedir éste método en la URL y otro parámetro que se va a llamar “method”, para indicarle a través del recurso “RequestMethod.GET”, el metodo por el cuál va a cargar esta petición, e importamos su clase relacionada. Dentro del método “form()”, podemos devolver el valor de la forma “new ModelAndView()”, sin necesidad de utilizar una variable.



```
package controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
public class FormController {
    @RequestMapping(value="form.htm",method=RequestMethod.GET)
    public ModelAndView form(){
        return new ModelAndView();
    }
}
```

Para trabajar con los campos de un formulario, los haremos con la parte de Modelo, creamos una clase, de nombre “Persona”, en el paquete “modelos”. Borramos los comentarios, y generamos una entidad, declarando una variable privada de tipo “int” para el campo “id”, dos variables privadas de tipo “String” para los campos “nombre” y “correo” y otra variable privada de tipo “int” para el campo “edad”.



```
package modelos;
public class Persona {
    private int id;
    private String nombre;
    private String correo;
    private int edad;
}
```

Con “Insert Code”, generamos un constructor vacío, con todos los métodos menos “id”, y los “Getters y Setters”.

```

package modelos;

public class Persona {
    private int id;
    private String nombre;
    private String correo;
    private int edad;

    public Persona() {
    }

    public Persona(int id, String nombre, String correo, int edad) {
        this.nombre = nombre;
        this.correo = correo;
        this.edad = edad;
    }

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getCorreo() {
        return correo;
    }
    public void setCorreo(String correo) {
        this.correo = correo;
    }
    public int getEdad() {
        return edad;
    }
    public void setEdad(int edad) {
        this.edad = edad;
    }
}

<bean name="indexController"
      class="org.springframework.web.servlet.mvc.ParameterizableViewController" />
<bean name="homeController" class="controller.HomeController" />
<bean name="formController" class="controller.FormController" />
</beans>
<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="index.htm">indexController</prop>
            <prop key="home.htm">homeController</prop>
            <prop key="contacto.htm">homeController</prop>
            <prop key="form.htm">formController</prop>
        </props>
    </property>
</bean>

```

Ahora creamos el fichero “**form.jsp**”, copiándolo de cualquiera de los otros JSP, y lo modificamos.

Al principio tenemos que usar un **taglib** de Spring para poder usar tags de formularios en este frameworks.

Entre “**<h1>**” ponemos “**Formulario Spring MVC**”.

Usamos un tag “**<form:form>**” para definir el formulario. Entre “**<p>**”, incluimos un tag “**<form:label>**” con texto “**Nombre**”, y otro “**<form:input>**”, los dos con el atributo “**path**” a “**nombre**”.

Hacemos lo mismo para los otros dos campos, “**correo**” con texto “**E-mail**”, y “**edad**” con texto “**Edad**”.

Terminamos creando un botón submit con “**<form:button>**”, de texto “**Enviar**”, y lo ejecutamos.

Volvemos al controlador para completar el “**return**” que tiene que devolver, añadiendo parámetros a la instancia de **ModelAndView**. Primero tendrá el nombre del formulario, “**form**”, después “**command**”, para informar de la clase asociada que se pasará como tercer parámetro, y que será una instancia de la clase **Persona**, y que importaremos.

```

package controller;

import modelos.Persona;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

public class FormController {
    @RequestMapping(value="form.htm",method=RequestMethod.GET)
    public ModelAndView form(){
        return new ModelAndView("form","command",new Persona());
    }
}

```

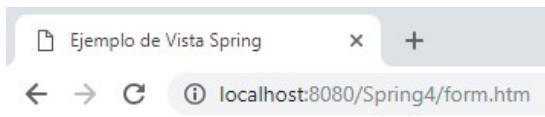
Ahora tenemos que informar sobre este controlador en el “**dispatcher-servlet.xml**”. Nos iremos al final del fichero para crear un objeto de “**<beans>**”. Con el parámetro “**name**” le indicamos el nombre de nuestro controlador, “**formController**” y con el atributo “**class**” le indicamos su ruta, “**controller. FormController**”.

También informamos de la vista, más arriba, entre las etiquetas “**<props>**”, introducimos una etiqueta “**<prop>**”, y con el parámetro “**key**” le indicamos que para “**form.htm**” se busque ese recurso en el controlador “**formController**”.

```

<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Ejemplo de Vista Spring</title>
    </head>
    <body>
        <h1>Formulario Spring MVC</h1>
        <form:form>
            <p>
                <form:label path="nombre">Nombre</form:label>
                <form:input path="nombre"></form:input>
            </p>
            <p>
                <form:label path="correo">E-mail</form:label>
                <form:input path="correo"></form:input>
            </p>
            <p>
                <form:label path="edad">Edad</form:label>
                <form:input path="edad"></form:input>
            </p>
            <form:button>Enviar</form:button>
        </form:form>
    </body>
</html>

```



## Formulario Spring MVC

Nombre

E-mail

Edad

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Ejemplo de Vista Spring</title>
  </head>
  <body>
    <h1>Formulario Spring MVC</h1>
    <form id="command" action="/Spring4/form.htm" method="post">
      <p>
        <label for="nombre">Nombre</label>
        <input id="nombre" name="nombre" type="text" value="" />
      </p>
      <p>
        <label for="correo">E-mail</label>
        <input id="correo" name="correo" type="text" value="" />
      </p>
      <p> <label for="edad">Edad</label>
        <input id="edad" name="edad" type="text" value="0"/>
      </p>
      <button type="submit" value="Submit">Enviar</button>
    </form>
  </body>
</html>
```

Vemos en el código fuente como al presionar el botón “Enviar” se va a redireccionar al mismo método.



## HTTP Status 405 - Request method &#39;POST&#39; not supported

**type** Status report  
**message**Request method &#39;POST&#39; not supported  
**description**The specified HTTP method is not allowed for the requested resource.

## GlassFish Server Open Source Edition 5.0

Volvemos al controlador “FormController” para procesar esto, y la forma más usual es procesar la información a través de un método de tipo texto.

Creamos un método de tipo String, que también podemos llamarle “**form()**”, utilizamos la anotación “**@RequestMapping**”, igual que el método anterior, pero esta vez, utilizamos “**POST**” para recibir nuestro formulario. Al método “**form()**”, le pasaremos por parámetros dos valores, un objeto “**per**” de tipo “**Persona**” y otro objeto de Spring “**model**” de tipo “**ModelMap**”, para procesar mapeos de modelos con una variante de  **ModelAndView**, y que tendremos que añadir su clase correspondiente. Utilizamos el atributo “**addAttribute**” de “**model**” para pasarle los valores, primero el campo, “**nombre**”, y luego el método “**getNombre()**” de la instancia “**per**”, y repetimos esta línea para los otros campos “**correo**” y “**edad**”.

```
package controller;

import modelos.Persona;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

public class FormController {
    @RequestMapping(value="form.htm",method=RequestMethod.GET)
    public ModelAndView form(){
        return new ModelAndView("form","command",new Persona());
    }

    @RequestMapping(value="form.htm",method=RequestMethod.POST)
    public String form(Persona per,ModelMap model){
        model.addAttribute("nombre", per.getNombre());
        model.addAttribute("correo", per.getCorreo());
        model.addAttribute("edad", per.getEdad());
        return "enviado";
    }
}
```

Terminamos devolviendo el nombre de la vista que va a contener estos valores, por ejemplo “**enviado**”.

Creamos el fichero “**enviado.jsp**”, copiándolo de otro JSP, y entre los “**<h1>**” ponemos “**Envío de formulario**”. Añadimos al principio el “**taglib**” de JSTL, y en una lista, “**<ul>**”, mostramos cada campo en un ítem “**<li>**”, comenzando por el texto “**Nombre:**” y usamos “**<c:out>**” con el valor “ **\${nombre}**”. Copiamos dos veces esta línea, y modificamos con el texto “**E-mail:**”, valor “ **\${correo}**” y con el texto “**Edad:**”, valor “ **\${edad}**”.

Ahora ya podemos rellenar el formulario y al darle el botón “Enviar” se mostrarán los valores introducidos.

```

<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Ejemplo de Vista Spring</title>
    </head>
    <body>
        <h1>Envío de formulario</h1>
        <ul>
            <li>Nombre : <c:out value="${nombre}" /></li>
            <li>E-Mail : <c:out value="${correo}" /></li>
            <li>Edad : <c:out value="${edad}" /></li>
        </ul>
    </body>
</html>

```

Vemos que no se visualiza correctamente los valores, y para solucionar esto, vamos a añadir un filtro en el fichero “**web.xml**”, después del “**</welcome-file-list>**”, tal y como se muestra a continuación. Los filtros son interceptores que lo que hacen es recoger actividades desde las distintas peticiones que nosotros vamos a realizar.

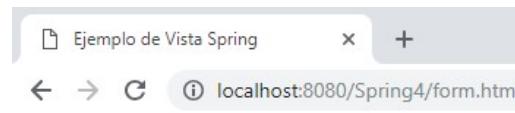
```

</welcome-file-list>
<filter>
    <filter-name>encodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <param-name>forceEnding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>encodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

```

Básicamente, le vamos a indicar que se fuerce el uso del cotejamiento UTF-8 con el primer filtro, y con el segundo, nos aseguramos de que todo cargue de manera correcta.

Después actualizamos para comprobar que se ve correctamente.



## Envío de formulario

- Nombre : Luis Estañ
- E-Mail : lestan@iesdoctorbalmis.com
- Edad : 49



## Envío de formulario

- Nombre : Luis Estañ
- E-Mail : lestan@iesdoctorbalmis.com
- Edad : 49

## 5 Validación de Formularios.

Continuamos copiando el proyecto anterior, “**Spring4**”, con el nombre “**Spring5**” para realizar modificaciones.

Vamos a mejorar la presentación de las vistas, utilizando Bootstrap desde su dirección web siguiente.

<https://getbootstrap.com/docs/3.3/getting-started/>

Modificamos la vista “**form.jsp**” incluyendo antes de cerrar la etiqueta “**</head>**” el enlace para acceder a su CSS.

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" >
```

Después de la etiqueta “**<body>**”, incluimos dos “**<div>**” con clases de Bootstrap, “**container**” y “**row**”.

A la etiqueta “**<form:form>**” le añadimos el atributo “**method**” con el valor de “**post**” y el atributo “**commandName**” con el valor “**persona**”, que es la instancia de la clase “**Persona**”.

Para cada campo, añadimos la clase de Bootstrap “**form-group**”, y para cada “**<form:input>**” el atributo “**cssClass**” con la clase de Bootstrap “**form-control**”

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Ejemplo de Vista Spring</title>
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" >
    </head>
    <body>
        <div class="container">
            <div class="row">
                <h1>Formulario Spring MVC</h1>
                <form:form method="post" commandName="persona">
                    <div class="form-group">
                        <form:label path="nombre">Nombre</form:label>
                        <form:input path="nombre" cssClass="form-control"></form:input>
                    </div>
                    <div class="form-group">
                        <form:label path="correo">E-mail</form:label>
                        <form:input path="correo" cssClass="form-control"></form:input>
                    </div>
                    <div class="form-group">
                        <form:label path="edad">Edad</form:label>
                        <form:input path="edad" cssClass="form-control"></form:input>
                    </div>
                    <form:button >Enviar</form:button>
                </form:form>
            </div>
        </div>
    </body>
</html>
```

Modificamos también el método “**form()**” de tipo “**ModelAndView**” para adaptarlo a las nuevas modificaciones.

```
public class FormController {
    @RequestMapping(value="form.htm",method=RequestMethod.GET)
    public ModelAndView form(){
        ModelAndView mav= new ModelAndView();
        mav.setViewName("form");
        mav.addObject("persona", new Persona());
        return mav;
    }
}
```

Vamos a cambiar el fichero “**redirect.jsp**” para que cargue directamente el formulario.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<% response.sendRedirect("form.htm"); %>
```

Probamos a ejecutar el proyecto, para comprobar si se visualiza el formulario aplicando las clases de Bootstrap.

Vamos a modificar en el “**FormController**” el método “**form**” de tipo “**String**”, ya que ahora, ademas de procesar el formulario, necesitamos validararlo. Para eso cambiamos el tipo a “**ModelAttribute**”, le pasamos varios parámetros, borramos el contenido, le añadimos las clases necesarias y borramos la que no necesitamos.

Para validar la información introducida en el formulario, vamos a crear una nueva clase que contenga nuestros métodos de validación. Señalamos “**Source Package**”, con botón derecho “**New**” y “**Java Class**” para crear un controlador, de nombre “**PersonaValidar**”, en el paquete “**modelos**”.

Utilizamos “**implements**” para indicarle la implementamos los métodos, necesarios por e

```
public class PersonaValidar implements Validator {  
     Implement all abstract methods  
     Make class PersonaValidar abstract
```

Con el método “**supports**”, realizaremos la comunicación de este objeto validador con la clase que queremos validar.

Con el método “**validate**”, agregamos todos los métodos de validación que necesitemos.

Completamos los métodos según se muestra a continuación.

```
package modelos;  
  
import org.springframework.validation.Errors;  
import org.springframework.validation.ValidationUtils;  
import org.springframework.validation.Validator;  
  
public class PersonaValidar implements Validator {  
  
    @Override  
    public boolean supports(Class<?> type) {  
        return Persona.class.isAssignableFrom(type);  
    }  
  
    @Override  
    public void validate(Object o, Errors errors) {  
        Persona persona=(Persona)o;  
        ValidationUtils.rejectIfEmptyOrWhitespace(errors,  
            "nombre",  
            "required.nombre",  
            "El campo Nombre es obligatorio");  
        ValidationUtils.rejectIfEmptyOrWhitespace(errors,  
            "correo",  
            "required.correo",  
            "El campo E-mail es obligatorio");  
        if (persona.getEdad()<18) errors.rejectValue("edad",  
            "edad.incorrect",  
            "El campo Edad no puede ser menor a 18");  
    }  
}
```

```
package controller;  
  
import modelos.Persona;  
import org.springframework.validation.BindingResult;  
import org.springframework.web.bind.annotation.ModelAttribute;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
import org.springframework.web.bind.support.SessionStatus;  
import org.springframework.web.servlet.ModelAndView;  
  
public class FormController {  
    @RequestMapping(value="form.htm",method=RequestMethod.GET)  
    public ModelAndView form(){  
        ModelAndView mav= new ModelAndView();  
        mav.setViewName("form");  
        mav.addObject("persona", new Persona());  
        return mav;  
    }  
  
    @RequestMapping(value="form.htm",method=RequestMethod.POST)  
    public ModelAndView form(@ModelAttribute("persona") Persona persona,  
        BindingResult result,  
        SessionStatus status){  
        
```

Volvemos a la clase “**FormController**” para completar el método utilizando ésta última clase, “**PersonaValidar**”. Creamos un atributo de tipo “**PersonaValidar**” llamado “**personaValidar**”, y creamos un constructor vacío, donde hacemos que “**personaValidar**” pase a ser un objeto de validación, y añadimos su clase.

Completamos el método para que la validación se pueda completar de forma correcta con el atributo “**validate**” de “**personaValidar**”, creamos la instancia de “**ModelAndView**”, y le añadimos un condicional para indicarle que hacer si se cumple un error de validación, que será volver a pedir los datos, y en el caso de que no haya ningún error y todo sea correcto, mostrará los valores introducidos.

Probamos la ejecución, y vemos que el formulario tiene el formato de Bootstrap, y que si provocamos error dejando campos vacíos, vuelve a mostrar el formulario, sin mostrar ningún mensaje.

Para que se muestren los mensajes de error utilizamos “**<form:errors>**” y tenemos dos opciones.

- Podemos agrupar todos los mensajes en un “**<div>**”, cada uno en una línea y con color destacado.
- Incluir cada mensaje junto al campo correspondiente.

Probaremos las dos formas. Comprobamos a enviar el formulario vacío para mostrar los mensajes.

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Example de Vista Spring</title>
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" >
    </head>
    <body>
        <div class="container">
            <div class="row">
                <h1>Formulario Spring MVC</h1>
                <form:form method="post" commandName="persona">
                    <form:errors path="" element="div" cssClass="alert alert-danger"></form:errors>
                    <div class="form-group">
                        <form:label path="nombre">Nombre</form:label>
                        <form:input path="nombre" cssClass="form-control"></form:input>
                        <form:errors path="nombre"></form:errors>
                    </div>
                    <div class="form-group">
                        <form:label path="correo">E-mail</form:label>
                        <form:input path="correo" cssClass="form-control"></form:input>
                        <form:errors path="correo"></form:errors>
                    </div>
                    <div class="form-group">
                        <form:label path="edad">Edad</form:label>
                        <form:input path="edad" cssClass="form-control"></form:input>
                        <form:errors path="edad"></form:errors>
                    </div>
                    <input class="btn btn-primary" type="submit" value="Submit">
                </form:form>
            </div>
        </div>
    </body>
</html>
```

Para terminar, vamos a ver como validar que el correo sea válido con las siguientes líneas.

```
private static final String EMAIL_PATTERN = "^[_A-Za-z0-9-\\+]+(\\.[_A-Za-z0-9-]+)*@[
    [A-Za-z0-9-]+(\\.[A-Za-z0-9-]+)*(\\.[A-Za-z]{2,})$";
private Pattern pattern;
private Matcher matcher;
```

Incluimos las clases necesarias, y al final, añadimos el código de validación.

```
if (!(persona.getCorreo() != null && persona.getCorreo().isEmpty()))
{
    this.pattern = Pattern.compile(EMAIL_PATTERN);
    this.matcher = pattern.matcher(persona.getCorreo());
    if (!matcher.matches()) {
        errors.rejectValue("correo", "correo.incorrect",
            "El E-Mail "+persona.getCorreo()+" no es válido");
    }
}
```

```
public class FormController {
    private PersonaValidar personaValidar;

    public FormController() {
        this.personaValidar=new PersonaValidar();
    }

    @RequestMapping(value="form.htm",method=RequestMethod.POST)
    public ModelAndView form(@ModelAttribute("persona") Persona per,
        BindingResult result,
        SessionStatus status) {
        this.personaValidar.validate(per, result);
        ModelAndView mav=new ModelAndView();
        if(result.hasErrors()){
            // Error en los datos y volvemos al formulario
            mav.setViewName("form");
            mav.addObject("per", new Persona());
        }else{
            // Datos correctos y mostramos los resultados
            mav.setViewName("enviado");
            mav.addObject("nombre",per.getNombre());
            mav.addObject("correo",per.getCorreo());
            mav.addObject("edad",per.getEdad());
        }
        return mav;
    }
}
```

## Formulario Spring MVC

The screenshot shows a web form with three fields: Nombre, E-mail, and Edad. Each field has an associated error message below it:

- Nombre: El campo Nombre es obligatorio
- E-mail: El campo E-mail es obligatorio
- Edad: El campo Edad no puede ser menor a 18

The form fields are:

- Nombre:** An input field with the placeholder "Nombre".
- E-mail:** An input field with the placeholder "E-mail".
- Edad:** An input field with the value "0".

A "Submit" button is at the bottom of the form.

Vemos como quedaría el formulario y probamos la ejecución.

```
package modelos;

import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

public class PersonaValidar implements Validator {
    private static final String EMAIL_PATTERN = "^[_A-Za-z0-9-\\]+(\\.[_A-Za-z0-9-]+)*@[A-Za-z0-9-]+(\\. [A-Za-z0-9]+)*(\\. [A-Za-z]{2,})$";
    private Pattern pattern;
    private Matcher matcher;

    @Override
    public boolean supports(Class<?> type) {
        return Persona.class.isAssignableFrom(type);
    }

    @Override
    public void validate(Object o, Errors errors) {
        Persona persona=(Persona)o;
        ValidationUtils.rejectIfEmptyOrWhitespace(errors,
                "nombre",
                "required.nombre",
                "El campo Nombre es obligatorio");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors,
                "correo",
                "required.correo",
                "El campo E-mail es obligatorio");
        if (persona.getEdad()<18) errors.rejectValue("edad",
                "edad.incorrect",
                "El campo Edad no puede ser menor a 18");
        if (!(persona.getCorreo() != null && persona.getCorreo().isEmpty())) {
            this.pattern = Pattern.compile(EMAIL_PATTERN);
            this.matcher = pattern.matcher(persona.getCorreo());
            if (!matcher.matches()) errors.rejectValue("correo",
                    "correo.incorrect",
                    "El E-Mail "+persona.getCorreo()+" no es válido");
        }
    }
}
```

## Formulario Spring MVC

El E-Mail loquesea no es válido

Nombre

Luis Estañ

E-mail

loquesea

El E-Mail loquesea no es válido

Edad

49

Submit

## 6 Conexión a MySQL con Jdbc Template.

Vamos crear un proyecto desde “Nuevo Proyecto”, Categoría “Java Web”, y “Web Application”, de nombre “Spring6”. Elegimos de servidor “GlassFish Server 5” y en Frameworks, marcamos “Spring Web MVC”, dejando marcada la casilla de “Include JSTL”, para incluir también estas librerías.

Creamos un controlador, de nombre “HomeController”, y el paquete “com.controller”.

Creamos un método llamado “home” de tipo “ModelAndView” y la anotación “@RequestMapping(“home.htm”)”.

Dentro del método, creamos una instancia de “ ModelAndView” llamada “mav”. Despues utilizamos “mav.setViewName”, para indicarle el nombre de la vista asociada, que será “home”.

```
package com.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

public class HomeController {
    @RequestMapping("home.htm")
    public ModelAndView home(){
        ModelAndView mav=new ModelAndView();
        mav.setViewName("home");
        return mav;
    }
}
```

Por ultimo, utilizamos “return mav” para devolver la variable “mav”.

Ahora tenemos que informar sobre este controlador en el “dispatcher-servlet.xml”.

Al final del fichero creamos un objeto de “<beans>”. Con “name” le indicamos el nombre del controlador, “homeController” y con el atributo “class” la ruta, “com.controller.HomeController”. Más arriba, entre las etiquetas “<props>”, vamos a introducir una etiqueta “<prop>”, y con el parámetro “key” le indicamos que para “home.htm” se busque ese recurso en el controlador “homeController”.

```
<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="index.htm">indexController</prop>
            <prop key="home.htm">homeController</prop>
        </props>
    </property>
</bean>

<bean id="viewResolver"
    class="org.springframework.web.servlet.view.InternalResourceViewResolver"
    p:prefix="/WEB-INF/jsp/"
    p:suffix=".jsp" />

<!--
The index controller.
-->
<bean name="indexController"
    class="org.springframework.web.servlet.mvc.ParameterizableViewController"
    p:viewName="index" />
<bean name="homeController" class="com.controller.HomeController" />
```

Para terminar, creamos el fichero “home.jsp”, usando Bootstrap, para formato, botones e iconos.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Ejemplo de JdbcTemplate</title>
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" />
    </head>
    <body>
        <div class="container">
            <div class="row">
                <h1>Ejemplo de JdbcTemplate</h1>
                <p>
                    <a href="#" class="btn btn-success">
                        <span class="glyphicon glyphicon-plus" aria-hidden="true">
                            </span> Agregar
                    </a>
                </p>
            </div>
        </div>
    </body>
</html>
```

Vamos a cambiar el fichero “**redirect.jsp**” para que cargue directamente el recurso con el que trabajamos.

Podemos probar la ejecución de lo que tenemos hasta el momento.

Continuamos, creando una clase, de nombre “**Conejar**”, en el paquete “**com.modelos**”. En esta clase, vamos a crear un método de tipo “**DataSource**” para poder construir la interfaz de conexión. Creamos un objeto de tipo “**DataSource**” y le indicamos el driver que vamos a utilizar.

Ahora tendremos que añadir la librería, señalando “**Libraries**”, y con el menú contextual señalamos “**Agregar biblioteca**”. Elegimos “**Driver MySQL JDBC**” en la lista y pulsamos “**Add Library**”.

Vamos a utilizar la base de datos “**practicas**”, con la tabla “**usuarios**”, e iniciamos “**MySQL**” del “**XAMPP**”

The screenshot shows the MySQL Workbench interface. At the top, it says "Servidor: 127.0.0.1 » Base de datos: practicas ». Tabla: usuarios". Below this are tabs for "Examinar", "Estructura", "SQL", "Buscar", and "Inserta". The "Estructura de tabla" tab is selected. It displays the table structure with columns: #, Nombre, Tipo, Cotejamiento, Atributos, Nulo, and Predeter. There are three rows: 1. id int(11), 2. nombre varchar(100) utf8\_spanish\_ci, and 3. correo varchar(100) utf8\_spanish\_ci. Each row has a checkbox next to it.

Añadimos ahora todos los atributos de conexión, y con el return, devolvemos el “**DataSource**”.

Si actualizamos el navegador, no tendría que salir ningún error.

Continuamos en el “**HomeController**”.

Vamos a crear un atributo de tipo “**JdbcTemplate**” que llamaremos “**jdbcTemplate**”. Creamos un constructor vacío, y dentro creamos una instancia de un objeto de tipo “**Conejar**”, que usaremos para realizar la conexión.

Ahora crearemos un consulta sql y un objeto de tipo “**List**” al que le pasamos como parámetro la consulta “**sql**” de antes. Y pasamos el parámetro “**datos**” a la vista con el atributo “**addObject**”.

Tenemos que ir a la vista para mostrar los datos.

Continuamos en “**home.jsp**” creando una tabla, donde aplicaremos clases de Bootstrap para bordes, alternar filas diferentes y destacar al pasar el ratón. Rellenamos la cabecera con los textos de los campos, y al final, las acciones que podremos realizar para cada registro.

Probamos que se visualiza la cabecera de la tabla correctamente.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<% response.sendRedirect("home.htm"); %>
```

## Ejemplo de JdbcTemplate

[+ Agregar](#)

```
package com.modelos;

import org.springframework.jdbc.datasource.DriverManagerDataSource;

public class Conejar {
    public DriverManagerDataSource conectar(){
        DriverManagerDataSource dataSource=new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    }
}
```

The screenshot shows the XAMPP Control Panel v3.2.2. At the top, it says “XAMPP Control Panel v3.2.2 [ Compiled: Nov 12th 2015 ]”. Below this is a table titled “XAMPP Control Panel v3.2.2” with columns: Service, Module, PID(s), Port(s), and Actions. It lists two services: Apache (PID 2345, Start button) and MySQL (PID 2356, Stop button).

```
public DriverManagerDataSource conectar(){
    DriverManagerDataSource dataSource=new DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost/practicas");
    dataSource.setUsername("root");
    dataSource.setPassword("");
    return dataSource;
}
```

```
public HomeController() {
    Conejar con=new Conejar();
    this.jdbcTemplate=new JdbcTemplate(con.conectar());

    @RequestMapping("home.htm")
    public ModelAndView home(){
        ModelAndView mav=new ModelAndView();
        String sql="select * from usuarios";
        List datos=this.jdbcTemplate.queryForList(sql);
        mav.setViewName("home");
        mav.addObject("datos", datos);
        return mav;
    }
}
```

```
<table class="table table-bordered table-striped table-hover">
    <thead>
        <tr>
            <th>ID</th>
            <th>Nombre</th>
            <th>E-Mail</th>
            <th>Acciones</th>
        </tr>
    </thead>
</table>
```

# Ejemplo de JdbcTemplate

+ Agregar

ID	Nombre	E-Mail	Acciones
----	--------	--------	----------

Visualizaremos los datos usando el “taglib” de JSTL y completando la tabla en el “<tbody>”, recorriendo la lista que nosotros pasamos por referencia. Aquí indicamos el código completo.

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Ejemplo de JdbcTemplate</title>
        <link rel="stylesheet"
            href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" >
    </head>
    <body>
        <div class="container">
            <div class="row">
                <h1>Ejemplo de JdbcTemplate</h1>
                <p>
                    <a href="" class="btn btn-success">
                        <span class="glyphicon glyphicon-plus" aria-hidden="true">
                        </span> Agregar
                    </a>
                </p>
                <table class="table table-bordered table-striped table-hover">
                    <thead>
                        <tr>
                            <th>ID</th>
                            <th>Nombre</th>
                            <th>E-Mail</th>
                            <th>Acciones</th>
                        </tr>
                    </thead>
                    <tbody>
                        <c:forEach items="${datos}" var="dato">
                            <tr>
                                <td><c:out value="${dato.id}"></c:out></td>
                                <td><c:out value="${dato.nombre}"></c:out></td>
                                <td><c:out value="${dato.correo}"></c:out></td>
                                <td></td>
                            </tr>
                        </c:forEach>
                    </tbody>
                </table>
            </div>
        </div>
    </body>
</html>
```

Y si probamos el resultado, se vería como se muestra a continuación.

# Ejemplo de JdbcTemplate

+ Agregar

ID	Nombre	E-Mail	Acciones
1	Luis	lestan@iesdoctorbalmis.com	
2	Pablo	pablo@iesdoctorbalmis.com	

# 7 CRUD con MySQL y Jdbc Template.

Continuamos copiando el proyecto anterior, “Spring6”, con el nombre “Spring7”, donde realizaremos las modificaciones necesarias para que podamos agregar, modificar y borrar registros.

Desde la vista, en el fichero “home.jsp”, vamos a completar los botones para realizar el CRUD, comenzando por el de “Agregar”.

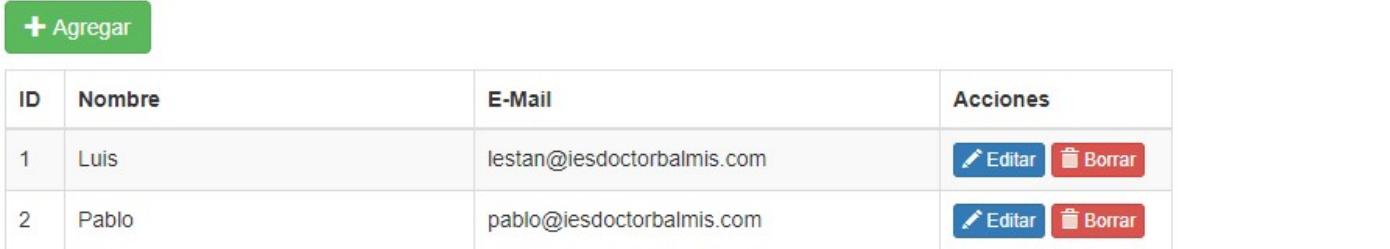
```
<a href="
```

Después copiamos todo el enlace anterior para preparar los botones de “Modificar” y “Borrar”, con las modificaciones necesarias. También ponemos un ancho de las cabeceras con valores porcentuales, de forma que se quede un diseño web adaptable.

```
<thead>
<tr>
    <th style="width:5%">ID</th>
    <th style="width:35%">Nombre</th>
    <th style="width:40%">E-Mail</th>
    <th style="width:20%">Acciones</th>
</tr>
</thead>

<tbody>
<tr>
    <td>1</td>
    <td>Luis</td>
    <td>lestan@iesdoctorbalmis.com</td>
    <td>
        <a href="
```

## Ejemplo de JdbcTemplate



ID	Nombre	E-Mail	Acciones
1	Luis	lestan@iesdoctorbalmis.com	<a href="#">Editar</a> <a href="#">Borrar</a>
2	Pablo	pablo@iesdoctorbalmis.com	<a href="#">Editar</a> <a href="#">Borrar</a>

Antes de crear un formulario, vamos a crear una clase java, del mismo nombre que la tabla que tenemos en la base de datos, “**Usuarios**” en nuestra capa de modelos, dentro del paquete “**com.modelos**”.

Crearemos un constructor vacío, otro constructor con todos los datos menos el “**id**”, otro constructor con todos los elementos, y por último, los “**Getters y Setters**”.

Después, en el mismo paquete “**com.modelos**”, crearemos una clase para validar los datos del formulario llamada “**UsuariosValidar**” y la completamos igual que se hizo cuando se vió la validación de formularios, en el proyecto “**Spring5**” con la clase “**PersonaValidar**”.

Utilizamos “**implements**” para indicarle la clase “**Validator**” de Spring, que importamos y después implementamos los métodos, necesarios por esta clase, “**supports**” y “**validate**”.

Copiamos el contenido de cada uno de estos métodos, cambiando el nombre de la clase, que este caso es “**Usuarios**”.

También copiamos y adaptamos la validación del “**E-Mail**”, creando las variables necesarias.

Se muestra a continuación, el resultado completo de como tiene que quedar.

```
package com.modelos;

public class Usuarios {
    private int id;
    private String nombre,correo;

    public Usuarios() {
    }
    public Usuarios(String nombre, String correo) {
        this.nombre = nombre;
        this.correo = correo;
    }
    public Usuarios(int id, String nombre, String correo) {
        this.id = id;
        this.nombre = nombre;
        this.correo = correo;
    }

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getCorreo() {
        return correo;
    }
    public void setCorreo(String correo) {
        this.correo = correo;
    }
}
```

```

package com.modelos;

import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

public class UsuariosValidar implements Validator{
    private static final String EMAIL_PATTERN = "^[_A-Za-z0-9-\\+]+(\\.[_A-Za-z0-9-]+)*@[A-Za-z0-9-]+(\\. [A-Za-z0-9-]+)*(\\. [A-Za-z]{2,})$";
    private Pattern pattern;
    private Matcher matcher;

    @Override
    public boolean supports(Class<?> type) {
        return Usuarios.class.isAssignableFrom(type);
    }

    @Override
    public void validate(Object o, Errors errors) {
        Usuarios usuarios=(Usuarios)o;
        ValidationUtils.rejectIfEmptyOrWhitespace(errors,
            "nombre",
            "required.nombre",
            "El campo Nombre es obligatorio");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors,
            "correo",
            "required.correo",
            "El campo E-mail es obligatorio");
        if (!(usuarios.getCorreo() != null && usuarios.getCorreo().isEmpty())) {
            this.pattern = Pattern.compile(EMAIL_PATTERN);
            this.matcher = pattern.matcher(usuarios.getCorreo());
            if (!matcher.matches()) errors.rejectValue("correo",
                "correo.incorrect",
                "El E-Mail "+usuarios.getCorreo()+" no es válido");
        }
    }
}

```

Vamos a ir creando la funcionalidad de “**Agregar**”, para eso crearemos un controlador llamado “**AddController**”, en el paquete “**com.controller**”.

Le añadimos las anotaciones “**@Controller**” y “**@RequestMapping(“add.htm”)**”. Generamos las instancias de “**UsuarioValidar**” y de “**JdbcTemplate**”. Creamos el constructor vacío, que va a tener la inicialización de “**UsuariosValidar()**”, la instancia de “**Conectar()**” y una nueva instancia de la clase “**JdbcTemplate**”.

Creamos el método “**form()**” de tipo “**ModelAndView**” para el formulario y añadimos “**@RequestMapping(method=RequestMethod.GET)**”.

Dentro del método, creamos una instancia de “**ModelAndView**” llamada “**mav**”. Después utilizamos “**mav.setViewName**”, para indicarle el nombre de la vista asociada, que será “**add**”. A través del atributo “ **addObject**” le pasamos una nueva instancia de la clase “**Usuarios**”, importando su clase. Por último, utilizamos “**return mav**” para devolver la variable “**mav**”.

Creamos el archivo “**add.jsp**” que contendrá el formulario, que podemos copiar y adaptar del fichero “**form.jsp**” del proyecto “**Spring5**”, sustituyendo la referencia “**persona**” por “**usuarios**”, eliminando el campo “**edad**”, y cambiar el título por “**Agregar usuario**”. También utilizaremos la librería de “**JSTL**”, y un componente Bootstrap llamado “**Breadcrumbs**”, comúnmente “**migas de pan**”, para que muestre la navegación jerárquica de la aplicación, e incluiremos todo el formulario en un componente panel, de título “**Formulario para agregar usuario**”.

```

package com.controller;

import com.modelos.Coneectar;
import com.modelos.Usuarios;
import com.modelos.UsuariosValidar;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping("add.htm")
public class AddController {
    UsuariosValidar usuariosValidar;
    private JdbcTemplate jdbcTemplate;

    public AddController() {
        this.usuariosValidar=new UsuariosValidar();
        Coneectar con=new Coneectar();
        this.jdbcTemplate=new JdbcTemplate(con.conectar());
    }

    @RequestMapping(method=RequestMethod.GET)
    public ModelAndView form(){
        ModelAndView mav=new ModelAndView();
        mav.setViewName("add");
        mav.addObject("usuarios", new Usuarios());
        return mav;
    }
}

```

```

<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@page contentType="text/html" pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Ejemplo de Vista Spring</title>
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" >
    </head>
    <body>
        <div class="container">
            <ol class="breadcrumb">
                <li><a href=<c:url value="/home.htm" />>Listado de usuarios</a></li>
                <li class="active">Agregar</li>
            </ol>
            <div class="panel panel-primary">
                <div class="panel-heading">Formulario para agregar usuario</div>
                <div class="panel-body">
                    <h1>Agregar usuario</h1>
                    <form:form method="post" commandName="usuarios">
                        <form:errors path="*" element="div" cssClass="alert alert-danger"></form:errors>
                        <div class="form-group">
                            <form:label path="nombre">Nombre</form:label>
                            <form:input path="nombre" cssClass="form-control"></form:input>
                        </div>
                        <div class="form-group">
                            <form:label path="correo">E-mail</form:label>
                            <form:input path="correo" cssClass="form-control"></form:input>
                        </div>
                        <input class="btn btn-primary" type="submit" value="Enviar">
                    </form:form>
                </div>
            </div>
        </div>
    </body>
</html>

```

Nos falta añadir el nuevo controlador, “**addController**” en el “**dispatcher-servlet.xml**”, y su vista.

```

<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="index.htm">indexController</prop>
            <prop key="home.htm">homeController</prop>
            <prop key="add.htm">addController</prop>
        </props>
    </property>
</bean>

<bean name="indexController"
      class="org.springframework.web.servlet.mvc.ParameterizableViewController"
      p:viewName="index" />
<bean name="homeController" class="com.controller.HomeController" />
<bean name="addController" class="com.controller.AddController" />

```

Probamos su ejecución, y que al pulsar el botón de “**Agregar**”, se va a mostrar el formulario.

Al principio, de la página, se puede ver el enlace que permite volver a la página principal.

The screenshot shows a web application interface. At the top, there's a navigation bar with the text "Listado de usuarios / Agregar". Below this, a blue header bar contains the text "Formulario para agregar usuario". Underneath the header, the main content area has a sub-header "Agregar usuario". There are two input fields: one for "Nombre" and one for "E-mail", both currently empty. At the bottom of the form is a blue "Enviar" button.

Nos falta aplicar la respuesta, para cuando envíemos el formulario, y es lo que vamos a realizar a continuación.

En el controlador “**AddController**” añadimos un método “**form()**”, de tipo “**ModelAndView**”, como el anterior, pero esta vez, con la anotación “**@RequestMapping(method=RequestMethod.POST)**”. Le pasamos tres parámetros al método, el primero es un “**@ModelAttribute**” asociandolo a la clase “**Usuarios**”, el segundo es un “**BindingResult**”, y el último un “**SessionStatus**”. Después, declaramos el método “**validate**”, asociado a la clase “**usuariosValidar**”, para ejecutar las validaciones. A continuación preguntamos si hay algún error con “**result.hasErrors()**”, en caso afirmativo, volvemos a procesar el formulario y sino usamos el objeto “**jdbcTemplate**” con su método “**update**”, que nos va a permitir realizar operaciones de insertar, modificar y borrar. El primer parámetro será la consulta “**sql**” que queremos realizar, y en el segundo parámetro se indicará cada uno de los campos, a los que hace referencia cada comodín del parámetro anterior. Por último, devolvemos un “ **ModelAndView**” redireccionando hasta el mismo listado, de “**/home.htm**”

```
@RequestMapping(method=RequestMethod.POST)
public ModelAndView form(@ModelAttribute("usuarios") Usuarios usuario,
                         BindingResult result,
                         SessionStatus status){
    this.usuariosValidar.validate(usuario, result);
    if(result.hasErrors()){
        ModelAndView mav=new ModelAndView();
        mav.setViewName("add");
        mav.addObject("usuarios", new Usuarios());
        return mav;
    }else{
        this.jdbcTemplate.update("insert into usuarios (nombre,correo) values (?,?)",
                               usuario.getNombre(),usuario.getCorreo());
        return new ModelAndView("redirect:/home.htm");
    }
}
```

Podemos probar a enviar un formulario vacío para comprobar si gestiona bien el error y ha añadir un nuevo usuario con datos correctos, para comprobar si se añade correctamente

Listado de usuarios / Agregar

### Formulario para agregar usuario

## Agregar usuario

El campo Nombre es obligatorio  
El campo E-mail es obligatorio

Nombre

E-mail

Enviar

Listado de usuarios / Agregar

### Formulario para agregar usuario

## Agregar usuario

Nombre

Mº Jesús García

E-mail

xusa@iesdoctorbalmis.com

Enviar

En la página principal, comprobamos que el nombre con tildes no se muestra correctamente.

## Ejemplo de JdbcTemplate

[+ Agregar](#)

ID	Nombre	E-Mail
1	Luis	lestan@iesdoctorbalmis.c
2	Pablo	pablo@iesdoctorbalmis.cc
3	MÁS JesÁs GarcÁa	xusa@iesdoctorbalmis.co

## Ejemplo de JdbcTemplate

[+ Agregar](#)

ID	Nombre	E-Mail
1	Luis	lestan@iesdoctorbalmis.c
2	Pablo	pablo@iesdoctorbalmis.cc
3	MÁS JesÁs GarcÁa	xusa@iesdoctorbalmis.co
4	Javier Catalá	javicalata@iesdoctorbalmi

Solucionamos esto copiando el filtro en el fichero “**web.xml**” que utilizamos en el proyecto de validación de formularios “**Spring5**”. Añadimos otro registro para comprobar el resultado.

Continuamos con la funcionalidad de “**Editar**” y creamos un nuevo controlador “**EditController**” en el paquete “**com.controller**”. Copiamos de “**AddController**” su contenido realizando modificaciones. Cambiamos “**AddController**” por “**EditController**”, y la vista “**add**” por “**edit**”.

Para que nos muestre los datos asociados al “**id**” que viene en la “**URL**”, al método “**GET**” como parámetro se le va a pasar un objeto de tipo “**HttpServletRequest**” y nombre, “**request**”, importando su clase. Declaramos la variable “**id**”, y le hacemos un “**parseInt**” para pasar su valor a una consulta sql. Se define el objeto “**datos**” que va a ser de tipo “**Usuarios**” y se le pasa el valor de “**id**” a través del método “**selectUsuario**” que se añadirá al final. Por el atributo “ **addObject**” le pasamos sus datos relacionados al “**id**” que se le ha pasado por url.

En el método “**POST**”, tendremos que añadir al parámetro “**form**” un objeto “**HttpServletRequest**” de nombre, “**request**”. Declaramos otra vez la variable “**id**”.

Si hay errores, se define el objeto “**datos**” que va a ser de tipo “**Usuarios**” y por el atributo “ **addObject**” le pasamos sus datos relacionados al “**id**” que se le ha pasado por url, para que los muestre al volver a cargar el formulario.

Si los datos son correctos, usamos el objeto “**jdbcTemplate**” con su método “**update**”, para realizar la consulta sql de actualización.

En el método “**selectUsuario**” se define un elemento de tipo final llamado “**user**” de la clase “**Usuarios**”. Luego se genera la query donde “**id**” es igual al “**id**” que se le pasa como parámetro. Después se devuelve un objeto de tipo “**Usuarios**” utilizando la clase “**jdbcTemplate**” y el método “**query**”, que permite procesar los datos a través de un método “**ResultSet**” y devolviendo cada uno de los campos del registro.

```

package com.controller;

import com.modelos.Conectar;
import com.modelos.Usuarios;
import com.modelos.UsuariosValidar;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.servlet.http.HttpServletRequest;
import org.springframework.dao.DataAccessViolationException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.support.SessionStatus;
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping("edit.htm")
public class EditController {
    UsuariosValidar usuariosValidar;
    private JdbcTemplate jdbcTemplate;

    public EditController() {
        this.usuariosValidar=new UsuariosValidar();
        Conectar con=new Conectar();
        this.jdbcTemplate=new JdbcTemplate(con.conectar());
    }

    @RequestMapping(method=RequestMethod.GET)
    public ModelAndView form(HttpServletRequest request) {
        ModelAndView mav=new ModelAndView();
        int id=Integer.parseInt(request.getParameter("id"));
        Usuarios datos=this.selectUsuario(id);
        mav.setViewName("edit");
        mav.addObject("usuarios", new Usuarios(id,datos.getNombre(),datos.getCorreo()));
        return mav;
    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView form(@ModelAttribute("usuarios") Usuarios usuario,
                           BindingResult result,
                           SessionStatus status,
                           HttpServletRequest request){
        this.usuariosValidar.validate(usuario, result);
        int id=Integer.parseInt(request.getParameter("id"));
        if(result.hasErrors()){
            ModelAndView mav=new ModelAndView();
            Usuarios datos=this.selectUsuario(id);
            mav.setViewName("edit");
            mav.addObject("usuarios", new Usuarios(id,datos.getNombre(),datos.getCorreo()));
            return mav;
        }else{
            this.jdbcTemplate.update("update usuarios set nombre=?,correo=? where id=?",
                                     usuario.getNombre(),usuario.getCorreo(),id);
            return new ModelAndView("redirect:/home.htm");
        }
    }

    public Usuarios selectUsuario(int id){
        final Usuarios user=new Usuarios();
        String sql="SELECT * FROM usuarios WHERE id='"+id+"'";
        return (Usuarios) jdbcTemplate.query(sql, new ResultSetExtractor<Usuarios>(){
            public Usuarios extractData(ResultSet rs) throws SQLException, DataAccessViolationException {
                if (rs.next()) {
                    user.setNombre(rs.getString("nombre"));
                    user.setCorreo(rs.getString("correo"));
                }
                return user;
            }
        });
    }
}

```

Vamos a crear el archivo “edit.jsp”, copiamos el contenido de “add.jsp” y realizamos los cambios oportunos.

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Ejemplo de Vista Spring</title>
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" >
    </head>
    <body>
        <div class="container">
            <ol class="breadcrumb">
                <li><a href="
```

Añadimos el nuevo controlador, “editController” en el “dispatcher-servlet.xml”, y su vista relacionada.

```
<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="index.htm">indexController</prop>
            <prop key="home.htm">homeController</prop>
            <prop key="add.htm">addController</prop>
            <prop key="edit.htm">editController</prop>
        </props>
    </property>
</bean>

<bean name="indexController"
      class="org.springframework.web.servlet.mvc.ParameterizableViewController"
      p:viewName="index" />
<bean name="homeController" class="com.controller.HomeController" />
<bean name="addController" class="com.controller.AddController" />
<bean name="editController" class="com.controller.EditController" />
```

Probamos la ejecución modificando el usuario que se guardó mal, de “M<sup>a</sup> Jesús García”.

Listado de usuarios / Editar

## Formulario para editar usuario

### Editar usuario

Nombre

E-mail

Enviar

# Ejemplo de JdbcTemplate

+ Agregar

ID	Nombre	E-Mail	Acciones
1	Luis	lestan@iesdoctorbalmis.com	<a href="#">Editar</a> <a href="#">Borrar</a>
2	Pablo	pablo@iesdoctorbalmis.com	<a href="#">Editar</a> <a href="#">Borrar</a>
3	Mª Jesús García	xusa@iesdoctorbalmis.com	<a href="#">Editar</a> <a href="#">Borrar</a>
4	Javier Catalá	javicatala@iesdoctorbalmis.com	<a href="#">Editar</a> <a href="#">Borrar</a>

Para la funcionalidad de “Borrar” creamos el controlador “DeleteController” en “com.controller” y copiamos el contenido del controlador anterior “EditController”, realizando los cambios oportunos.

```
package com.controller;

import com.modelos.Conectar;
import com.modelos.Usuarios;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.servlet.http.HttpServletRequest;
import org.springframework.dao.DataAccessViolationException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping("delete.htm")
public class DeleteController {
    private JdbcTemplate jdbcTemplate;

    public DeleteController() {
        Conectar con=new Conectar();
        this.jdbcTemplate=new JdbcTemplate(con.conectar());
    }

    @RequestMapping(method=RequestMethod.GET)
    public ModelAndView form(HttpServletRequest request){
        ModelAndView mav=new ModelAndView();
        int id=Integer.parseInt(request.getParameter("id"));
        Usuarios datos=this.selectUsuario(id);
        mav.setViewName("delete");
        mav.addObject("usuarios", new Usuarios(id,datos.getNombre(),datos.getCorreo()));
        mav.addObject("id", datos.getId());
        mav.addObject("nombre", datos.getNombre());
        mav.addObject("correo", datos.getCorreo());
        return mav;
    }

    @RequestMapping(method=RequestMethod.POST)
    public ModelAndView form(@ModelAttribute("usuarios") Usuarios usuario,
                           HttpServletRequest request){
        int id=Integer.parseInt(request.getParameter("id"));
        this.jdbcTemplate.update("delete from usuarios where id=?",id);
        return new ModelAndView("redirect:/home.htm");
    }

    public Usuarios selectUsuario(int id){
        final Usuarios user=new Usuarios();
        String sql="SELECT * FROM usuarios WHERE id='"+id+"'";
        return (Usuarios) jdbcTemplate.query(sql, new ResultSetExtractor<Usuarios>(){
            public Usuarios extractData(ResultSet rs) throws SQLException, DataAccessException {
                if (rs.next()){
                    user.setNombre(rs.getString("nombre"));
                    user.setCorreo(rs.getString("correo"));
                }
                return user;
            }
        });
    }
}
```

Añadimos el nuevo controlador, “deleteController” en el “dispatcher-servlet.xml”, y su vista relacionada.

```

<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="index.htm">indexController</prop>
            <prop key="home.htm">homeController</prop>
            <prop key="add.htm">addController</prop>
            <prop key="edit.htm">editController</prop>
            <prop key="delete.htm">deleteController</prop>
        </props>
    </property>
</bean>

<bean name="indexController"
      class="org.springframework.web.servlet.mvc.ParameterizableViewController"
      p:viewName="index" />
<bean name="homeController" class="com.controller.HomeController" />
<bean name="addController" class="com.controller.AddController" />
<bean name="editController" class="com.controller.EditController" />
<bean name="deleteController" class="com.controller.DeleteController" />

```

Creamos el fichero “**delete.jsp**” copiando el contenido del “**edit.jsp**” y modificando lo necesario. Sustituimos donde pone “**editar**” por “**borrar**”, cambiamos los “**<form:input>**” por “**<c:out>**” para mostrar los valores dentro de un “**<div>**” de la clase Bootstrap “**form-control**”, borramos el “**<form:errors>**” porque no lo necesitamos, y por último, que el botón submit sea de color rojo con el texto “**Borrar**”.

```

<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Ejemplo de Vista Spring</title>
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
    </head>
    <body>
        <div class="container">
            <ol class="breadcrumb">
                <li><a href="

```

Probamos a borrar un registro, por ejemplo el primero, y vemos que funciona correctamente.

Listado de usuarios / Borrar

Confirmación para borrar usuario

## Borrar usuario

Nombre

E-mail

**Borrar**

## Ejemplo de JdbcTemplate

ID	Nombre	E-Mail
2	Pablo	pablo@iesdoctorbalmis.co
3	Mª Jesús García	xusa@iesdoctorbalmis.co
4	Javier Catalá	javicalata@iesdoctorbalmi